

# A Note on Amortized Space Complexity

Aaron Potechin  
Institute for Advanced Study

November 22, 2016

## Abstract

In this note, we show that while almost all functions require exponential size branching programs to compute, for all functions  $f$  there is a branching program computing a large number of copies of  $f$  which has linear size per copy of  $f$ . We then discuss which non-monotone lower bound approaches for branching programs are ruled out by this result.

**Acknowledgement:** This work was supported by the Simons Foundation and the National Science Foundation. The author would like to thank Avi Wigderson for helpful conversations.

# 1 Introduction

An important limitation of branching programs is that we are not allowed to duplicate nodes. Thus, if we need a given computation multiple times, we generally have to recompute it each time. In this note, we consider the size of branching programs which compute multiple copies of the same function. We show that while almost all functions require exponential size branching programs to compute, for all functions  $f$ , if we compute  $f$  enough times, the branching program size needed per copy of  $f$  is only linear.

We then briefly discuss which lower bound methods are ruled out by this construction. In particular, we note that the current framework for proving lower bounds on monotone switching networks [1] cannot be extended to prove non-trivial non-monotone lower bounds unless it is significantly modified. We also note that our result is a constructive analogue of Razborov's result [3] that there is a linear upper bound on submodular complexity measures. However, this construction does not say anything about lower bounds based on counting functions such as Neciporuk's quadratic lower bound [2] or lower bounds based on communication complexity arguments.

## 2 Definitions

**Definition 2.1.** *We say that a branching program computes  $f$   $m$  times if the following is true:*

1. *The branching program has  $m$  start nodes  $s_1, \dots, s_m$ ,  $m$  accept nodes  $a_1, \dots, a_m$ , and  $m$  reject nodes  $r_1, \dots, r_m$*
2. *For all  $i$ , if the branching program starts at  $s_i$  then it will end at  $a_i$  if  $f(x) = 1$  and it will end at  $r_i$  if  $f(x) = 0$*

**Definition 2.2.**

1. *We define  $b_m(f)$  to be the minimal size of a branching program which computes  $f$   $m$  times.*
2. *We define  $b_{avg}(f) = \lim_{m \rightarrow \infty} \frac{b_m(f)}{m}$*

**Remark 2.3.** *Up to the issue of non-uniformity, the space needed to compute a function is proportional to the logarithm of the size of a branching program needed to compute that function. Thus, we can think of the (non-uniform) amortized space complexity of  $f$  as being proportional to  $\log(b_{avg}(f))$*

## 3 Upper bound on $b_{avg}(f)$

**Theorem 3.1.** *For all  $f$ ,  $b_{avg}(f) \leq 64n$ . In particular, for all  $f$ , taking  $m = 2^{2^n - 1}$ ,  $b_m(f) \leq 32n2^{2^n}$*

*Proof.* Our branching program has several parts. We first describe each of these parts and how we put them together and then we will describe how to construct each part. The first two parts are as follows:

1. A branching program which simultaneously identifies all functions  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  that have value 1 for a given  $x$ . More precisely, it has start nodes  $s_1, \dots, s_m$  where  $m = 2^{2^n-1}$  and has one end node  $t_f$  for each possible function  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ , with the guarantee that if  $f(x) = 1$  for a given  $f$  and  $x$  then there exists an  $i$  such that the branching program goes from  $s_i$  to  $t_f$  on input  $x$ .
2. A branching program which simultaneously computes all functions  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . More precisely, it has one start node  $s_f$  for each function  $f$  and has end nodes  $a_1, \dots, a_m$  and  $r_1, \dots, r_m$ , with the guarantee that for a given  $f$  and  $x$ , if  $f(x) = 1$  then the branching program goes from  $s_f$  to  $a_i$  for some  $i$  and if  $f(x) = 0$  then the branching program goes from  $s_f$  to  $r_i$  for some  $i$ .

If  $g$  is the function which we actually want to compute, we combine these two parts as follows. The first part gives us paths from  $\{s_i : i \in [1, m]\}$  to  $\{t_f : f(x) = 1\}$ . We now take each  $t_f$  from the first part and set it equal to  $s_{(f \wedge g) \vee (\neg f \wedge \neg g)}$  in the second part. Once we do this, if  $g(x) = 1$  then for all  $f$  such that  $f(x) = 1$ ,  $(f \wedge g) \vee (\neg f \wedge \neg g) = 1$  so we will have paths from  $\{t_f : f(x) = 1\} = \{s_{(f \wedge g) \vee (\neg f \wedge \neg g)} : f(x) = 1\}$  to  $\{a_i : i \in [1, m]\}$ . Following similar logic, if  $g(x) = 0$  then we will have paths from  $\{t_f : f(x) = 1\} = \{s_{(f \wedge g) \vee (\neg f \wedge \neg g)} : f(x) = 1\}$  to  $\{r_i : i \in [1, m]\}$ . Putting everything together, when  $g(x) = 1$  we will have paths from  $\{s_i : i \in [1, m]\}$  to  $\{a_i : i \in [1, m]\}$  and when  $g(x) = 0$  we will have paths from  $\{s_i : i \in [1, m]\}$  to  $\{r_i : i \in [1, m]\}$ .

However, these paths do not have to map  $s_i$  to  $a_i$  or  $r_i$ , they can permute the final destinations. To fix this, our final part will run the branching program we have so far in reverse. This fixes the permutation issue but gets us right back where we started! To avoid this, we have two copies of this final part, one applied to  $\{a_i : i \in [1, m]\}$  and one applied to  $\{r_i : i \in [1, m]\}$ . This separates the case when  $g(x) = 1$  and the case  $g(x) = 0$ , giving us our final branching program.

We now describe how to construct each part. For the first part, which simultaneously identifies the functions which have value 1 on input  $x$ , we have a layered branching program where at layer  $j$ , for each function  $f : \{0, 1\}^j \rightarrow \{0, 1\}$ , we have  $2^{2^n-2^j}$  nodes corresponding to  $f$ . We draw the arrows from level  $j-1$  to level  $j$  as follows. For a node corresponding to a function  $f : \{0, 1\}^{j-1} \rightarrow \{0, 1\}$ , we draw an arrow with label  $x_j = 1$  from it to a node corresponding to a function  $f' : \{0, 1\}^j \rightarrow \{0, 1\}$  such that  $f'(b_1, \dots, b_{j-1}, 1) = f(b_1, \dots, b_{j-1})$ . Similarly, we draw an arrow with label  $x_j = 0$  from it to a node corresponding to a function  $f' : \{0, 1\}^j \rightarrow \{0, 1\}$  such that  $f'(b_1, \dots, b_{j-1}, 0) = f(b_1, \dots, b_{j-1})$ . We make these choices arbitrarily but make sure that no two arrows with the same label have the same destination.

For the second part, which simultaneously computes each function, we have a layered branching program where at level  $j$ ,  $f : \{0, 1\}^{n-j} \rightarrow \{0, 1\}$ , we have  $2^{2^n-2^{n-j}}$  nodes corresponding to  $f$ . We draw the arrows from level  $n-j$  to level  $n-j+1$  as follows. For a node corresponding to a function  $f : \{0, 1\}^j \rightarrow \{0, 1\}$ , we draw an arrow with label  $x_j = 1$  from it to a node corresponding to the function  $f(b_1, \dots, b_{j-1}, 1)$  and draw an arrow with label  $x_j = 0$  from it to a node corresponding to the function  $f(b_1, \dots, b_{j-1}, 0)$ . Again, we make these choices arbitrarily but make sure that no two edges with the same label have the same destination.

For the final part, we note that because we made sure not to have any two edges with the same label have the same destination, it is easy to invert each individual layer and thus the branching program as a whole.  $\square$

## 4 Barrier for input-based bottleneck arguments

One way we could try to show lower bounds on branching programs is as follows. We could argue that for the given function  $f$  and a given branching program  $B$  computing  $f$ , for every YES input  $x$  the path that  $B$  takes on input  $x$  contains a vertex giving a lot of information about  $x$  and thus  $B$  must be large to accomodate all of the possible inputs. In this section, we observe that Theorem 3.1 rules out this kind of argument. In particular, we show the following.

**Lemma 4.1.** *Assume that we have a function  $f$  and a set of YES inputs  $I$  of  $f$ . If there is a criterion for assigning vertices of a branching program to inputs in  $I$  such that*

1. *For all  $x \in I$  and any path in a branching program from a start node to an accept node, there is a vertex  $v_x$  in this path which can be assigned to  $x$ .*
2. *For any vertex  $v$  in any branching program computing  $f$ ,  $v$  can be assigned to at most  $\frac{1}{S}$  of the inputs in  $I$*

*then  $b_m(f) \geq mS$*

*Proof.* Let  $B$  be a branching program computing  $f$   $m$  times. The total number of times a vertex in  $B$  is assigned to an input in  $I$  is at least  $m|I|$ . However, each vertex of the branching program can be assigned to at most  $\frac{|I|}{S}$  inputs in  $I$ , so the total number of times a vertex is assigned to an input in  $I$  is at most  $\frac{|V(B)| \cdot |I|}{S}$ , where  $|V(B)|$  is the size of the branching program. Thus,  $\frac{|V(B)| \cdot |I|}{S} \geq m|I|$  which implies that  $|V(B)| \geq mS$ , as needed.  $\square$

By Theorem 3.1, this implies that  $S \leq 64n$ , so no such argument can prove a superlinear lower bound. In fact, this is true even for oblivious read-twice branching programs.

## 5 Barrier for complexity measures

Another way we could try to lower bound branching program size is through a complexity measure on functions. It was shown by Razborov [3] that submodular complexity measures cannot have superlinear values, in this section we show that this is also true for a similar class of complexity measures suitable for lower bounding branching program size.

**Definition 5.1.** *We define a branching complexity measure  $\mu_b$  to be a measure on functions which satisfies the following properties*

1.  $\forall i, \mu_b(x_i) = \mu_b(\neg x_i) = 1$
2.  $\forall f, \mu_b(f) \geq 0$
3.  $\forall f, i, \mu_b(f \wedge x_i) + \mu_b(f \wedge \neg x_i) \leq \mu_b(f) + 2$
4.  $\forall f, g, \mu_b(f \vee g) \leq \mu_b(f) + \mu_b(g)$

**Definition 5.2.** *Given a node  $v$  in a branching program, define  $f_v(x)$  to be the function such that  $f_v(x)$  is 1 if there is a path from some start node to  $v$  on input  $x$  and 0 otherwise. Note that for any start node  $s$ ,  $f_s = 1$ .*

**Lemma 5.3.** *If  $\mu_b$  is a branching complexity measure then for any branching program, the number of non-end nodes which it contains is at least  $\frac{1}{2} (\sum_{t: t \text{ is an end node}} \mu_b(f_t) - \sum_{s: s \text{ is a start node}} \mu_b(f_s))$ .*

*Proof.* To see this, consider what happens to  $\sum_{t: t \text{ is an end node}} \mu_b(f_t) - \sum_{s: s \text{ is a start node}} \mu_b(f_s)$  as we construct the branching program. At the start, when we only have the start nodes and these are also our end nodes, this expression has value 0. Each time we merge end nodes together, this can only decrease this expression. Each time we branch off from an end node, making the current node a non-end node and creating two new end nodes, this expression increases by at most 2. Thus, the final value of this expression is at most twice the number of non-end nodes in the final branching program, as needed.  $\square$

**Corollary 5.4.** *For any branching complexity measure  $\mu_b$  and any function  $f$ ,  $\mu_b(f) \leq 130n$*

*Proof.* By Lemma 5.3 we have that for all  $m \geq 1$ ,  $\frac{m\mu_b(f) - m\mu_b(1)}{2} \leq m \cdot b_m(f)$ . Using Theorem 3.1 and noting that  $\mu_b(1) \leq 2$  we obtain that  $\mu_b(f) \leq 130n$ .  $\square$

Finally, we note that every submodular complexity measure  $\mu_s$  is a branching complexity measure, so Corollary 5.4 is a slight generalization of Razborov's result [3] (though with a worse constant).

**Definition 5.5.** *A submodular complexity measure  $\mu_s$  is a measure on functions which satisfies the following properties*

1.  $\forall i, \mu(x_i) = \mu(\neg x_i) = 1$
2.  $\forall f, \mu(f) \geq 0$
3.  $\forall f, g, \mu_s(f \vee g) + \mu_s(f \wedge g) \leq \mu_s(f) + \mu_s(g)$

**Lemma 5.6.** *Every submodular complexity measure  $\mu_s$  is a branching complexity measure.*

*Proof.* Note that

$$\mu_s(f \vee x_i) + \mu_s(f \wedge x_i) \leq \mu_s(f) + \mu_s(x_i)$$

and

$$\mu_s((f \vee x_i) \wedge \neg x_i) + \mu_s((f \vee x_i) \vee \neg x_i) = \mu_s(f \wedge \neg x_i) + \mu_s(1) \leq \mu_s(f \vee x_i) + \mu_s(\neg x_i)$$

Combining these two inequalities we obtain that

$$\mu_s(f \wedge \neg x_i) + \mu_s(1) + \mu_s(f \wedge x_i) \leq \mu_s(f) + \mu_s(x_i) + \mu_s(\neg x_i)$$

which implies that  $\mu_s(f \wedge \neg x_i) + \mu_s(f \wedge x_i) \leq \mu_s(f) + 2 - \mu_s(1) \leq \mu_s(f) + 2$ , as needed.  $\square$

## References

- [1] S. Chan. A. Potechin. Tight Bounds for Monotone Switching Networks via Fourier Analysis. Theory of Computing Volume 10 (2014) p.389-419
- [2] E. Neciporuk. On a Boolean function. Doklady of the Academy of Sciences of the USSR, 169 (4)
- [3] A. Razborov. On Submodular Complexity Measures. Proceedings of the London Mathematical Society Symposium on Boolean function complexity p.76 - 83.